Overview of Java 8 Streams (Part 5)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand the structure & functionality of Java 8 streams, e.g.,
  - Fundamentals of streams
  - Common stream aggregate operations
  - “Splittable iterators” (Spliterators)
  - Terminating a stream
  - Implementing non-concurrent collectors for sequential streams

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html
Implementing a Non-Concurrent Collector
Implementing a Non-Concurrent Collector

- A collector is used to terminate a stream

```java
void runCollectToList() {
    List<String> characters = Arrays.asList("horatio", "laertes", "Hamlet, ...");

    List<String> results =
        characters.stream()
            .filter(s ->
                toLowerCase(...) == 'h')
            .map(this::capitalize)
            .sorted()
            .collect(toList()); ...
}
```

Collect the results into a ArrayList

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)
Implementing a Non-Concurrent Collector

- A collector is used to terminate a stream
- Collector defines an interface whose implementations can accumulate input elements in a mutable result container

### Interface Collector<T,A,R>

**Type Parameters:**
- T - the type of input elements to the reduction operation
- A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)
- R - the result type of the reduction operation

```java
public interface Collector<T,A,R>

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class Collectors provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)
Implementing a Non-Concurrent Collector

- Collector implementations can either be non-concurrent or concurrent based on their characteristics.

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html](docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html)

<table>
<thead>
<tr>
<th>Enum Constant Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enum Constants</td>
</tr>
<tr>
<td>Enum Constant and Description</td>
</tr>
<tr>
<td>CONCURRENT</td>
</tr>
<tr>
<td>Indicates that this collector is <em>concurrent</em>, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.</td>
</tr>
<tr>
<td>IDENTITY_FINISH</td>
</tr>
<tr>
<td>Indicates that the finisher function is the identity function and can be elided.</td>
</tr>
<tr>
<td>UNORDERED</td>
</tr>
<tr>
<td>Indicates that the collection operation does not commit to preserving the encounter order of input elements.</td>
</tr>
</tbody>
</table>
Implementing a Non-Concurrent Collector

- Collector implementations can either be non-concurrent or concurrent based on their characteristics.
- This distinction is only relevant for parallel streams.

```
filter(not(this::urlCached))
map(this::downloadImage)
flatMap(this::applyFilters)
collect(toList())
```
Collector implementations can either be non-concurrent or concurrent based on their characteristics.

- This distinction is only relevant for parallel streams.
- Our focus here is on non-concurrent collectors for sequential streams.

Non-concurrent & concurrent collectors for parallel streams are covered later.
Implementing a Non-Concurrent Collector

• A non-concurrent collector for a sequential stream simply accumulates elements into a mutable result container
Implementing a Non-Concurrent Collector

- A collector is essentially the inverse of a spliterator

![Diagram showing the relationship between a collector and a spliterator](image)
Implementing a Non-Concurrent Collector

- A collector is essentially the inverse of a spliterator.

A spliterator splits a single input source into a stream of elements.
Implementing a Non-Concurrent Collector

- A collector is essentially the inverse of a spliterator

A collector combines a stream of elements back into a single result
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types

```java
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
  - **T** – The type of objects available in the stream
    - e.g., Integer, String, etc.

```java
<<Java Interface>>

Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
  - T
  - A – The type of a mutable accumulator object for collection
    - e.g., List of T (implemented via ArrayList or LinkedList)

```
<<Java Interface>>
Collector<T>A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>
```
Implementing a Non-Concurrent Collector

- The Collector interface defines three generic types
  - T
  - A
  - R – The type of a final result
    - e.g., List of T
Implementing a Non-.Concurrent Collector

- Five methods are defined in the Collector interface

Again, this discussion assumes we’re implementing a non-concurrent collector.
Implementing a Non-Concurrent Collector

• Five methods are defined in the Collector interface

• **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
  - UNORDERED
  - The collector need not preserve the encounter order
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
  - UNORDERED
    - The collector need not preserve the encounter order

A collector may preserve encounter order if it incurs no additional overhead
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
- characteristics() – provides a stream with additional information used for internal optimizations, e.g.
  - UNORDERED
- IDENTITY_FINISH
  - The finisher() is the identity function so it can be a no-op
    - e.g., finisher() just returns null
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
  - CONCURRENT
    - The accumulator() method is called concurrently on the result container

The mutable result container must be synchronized!!
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTITY_FINISH
    - CONCURRENT
      - The accumulator() method is called concurrently on the result container

<<Java Interface>>

1. Collector<T,A,R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A,T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A,R>
- characteristics(): Set<Characteristics>

We’re focusing on a non-concurrent collector, which doesn’t enable CONCURRENT.
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
- `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.

```java
Set characteristics() {
    return Collections.unmodifiableSet
            (EnumSet.of(Collector.Characteristics.CONCURRENT,
                         Collector.Characteristics.UNORDERED,
                         Collector.Characteristics.IDENTITY_FINISH));
}
```

Any/all characteristics can be set using `EnumSet.of()`

See [docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html](docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html)
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()` – returns a supplier instance that acts as a factory to generate an empty result container, e.g.

```java
Supplier<List> supplier() {
    return ArrayList::new;
}
```
Implementing a Non-.Concurrent Collector

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - **accumulator()** – returns a bi-consumer that adds a new element to an existing result container, e.g.

```java
BiConsumer<List, Integer> accumulator() {
    return List::add;
}
```

A non-concurrent collector needs no synchronization
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - **combiner()** – returns a binary operator that merges two result containers together, e.g.

```java
BinaryOperator<List> combiner() {
    return (one, another) -> {
        one.addAll(another);
        return one;
    };
}
```

This combiner() will not be called for a sequential stream.
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
- finisher() – returns a function that converts the result container to final result type, e.g.
  - return Function.identity()
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - **finisher()** – returns a function that converts the result container to final result type, e.g.
    - `return Function.identity()`
    - `return null;`

Should be a no-op if IDENTITY_FINISH characteristic is set
Implementing a Non-Concurrent Collector

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - finisher() – returns a function that converts the result container to final result type, e.g.
    - return Function.identity()
    - return null;

Stream
  .generate(() ->
    makeBigFraction
    (new Random(), false))
  .limit(sMAX_FRACTIONS)
  .map(reduceAndMultiplyFraction)
  .collect(FuturesCollector.toFuture())
  .thenAccept
    (this::sortAndPrintList);

finisher() can also be much more interesting!

See Java8/ex19/src/main/java/utils/FuturesCollector.java
Collectors is a utility class whose factory methods create collectors for common collection types.

```
public final class Collectors extends Object

Implementations of Collector that implement various useful reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc.

The following are examples of using the predefined collectors to perform common mutable reduction tasks:
```

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html)
Implementing a Non-Concurrent Collector

Collectors is a utility class whose factory methods create collectors for common collection types.

A utility class is a final class having only static methods, no state, & a private constructor.

Implementing a Non-Concurrent Collector

- Collectors is a utility class whose factory methods create collectors for common collection types
- e.g., returns a Collector that accumulates input elements into a new (Array)List

```java
final class Collectors {
    ...
    public static <T> Collector
        <T, ?, List<T>>
        toList() {
            return new CollectorImpl<>((Supplier<List<T>>)()
                ArrayList::new,
                List::add,
                (left, right) -> {
                    left.addAll(right);
                    return left;
                },
                CH_ID);
        }
    ...
}
```

See docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#toList
Implementing a Non-Concurrent Collector

- CollectorImpl defines a simple (private) implementation class for a Collector

See openjdk/8-b132/java/util/stream.Collectors.java#Collectors.CollectorImpl
Implementing a Non-Concurrent Collector

- Collector.of() defines a simple (public) factory method that implements a Collector using (private) CollectorImpl

```java
interface Collector<T, A, R> {
    ...
    static<T, R> Collector<T, R, R> of
        (Supplier<R> supplier,
        BiConsumer<R, T> accumulator,
        BinaryOperator<R> combiner,
        Characteristics... chars) {
        ...
        return new Collectors
            .CollectorImpl<>
            (supplier,
            accumulator,
            combiner,
            chars);
    }
    ...
}
```

See openjdk/8-b132/java/util/stream.Collectors.java#Collectors.CollectorImpl
Implementing a Non-Concurrent Collector

- Collector.of() can also implement custom collectors that have pithy lambdas

```java
public String toString() {
    ...
    mList.stream()
        .collect(Collectors.of(() -> new StringJoiner("|")
            (j, r) -> j.add(r.toString()),
            StringJoiner::merge,
            StringJoiner::toString)));
    ...
}
```

SearchResults’s custom collector formats itself

See SimpleSearchStream/src/main/java/search/SearchResults.java
Implementing a Non-Concurrent Collector

- Complex custom collectors should implement the Collector interface

Java Interface:

```java
public interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```

Java Class:

```java
public class FuturesCollector<T> {
    FuturesCollector()
    Supplier<List<CompletableFuture<T>>> supplier();
    BiConsumer<List<CompletableFuture<T>>, CompletableFuture<T>> accumulator();
    BinaryOperator<List<CompletableFuture<T>>, CompletableFuture<T>> combiner();
    Function<List<CompletableFuture<T>>, CompletableFuture<List<T>>> finisher();
    Set<Characteristics> characteristics();
    Collector<CompletableFuture<T>, ?, CompletableFuture<List<T>>> toFuture();
}
```

See [Java8/ex19/src/main/java/utils/FuturesCollector.java](Java8/ex19/src/main/java/utils/FuturesCollector.java)
Implementing a Non-Concurrent Collector

• More information on implementing custom collectors is available online

See [www.youtube.com/watch?v=H7VbRz9aj7c](http://www.youtube.com/watch?v=H7VbRz9aj7c)
End of Overview of Java 8 Streams (Part 5)